HA^{2}lloc: Hardware-Assisted Secure Allocator

Orlando Arias[†], Dean Sullivan[‡], Yier Jin[‡]

{oarias,dean.sullivan}@knights.ucf.edu
yier.jin@ece.ufl.edu

[†]University of Central Florida
[‡]University of Florida

June 25, 2017

Motivation

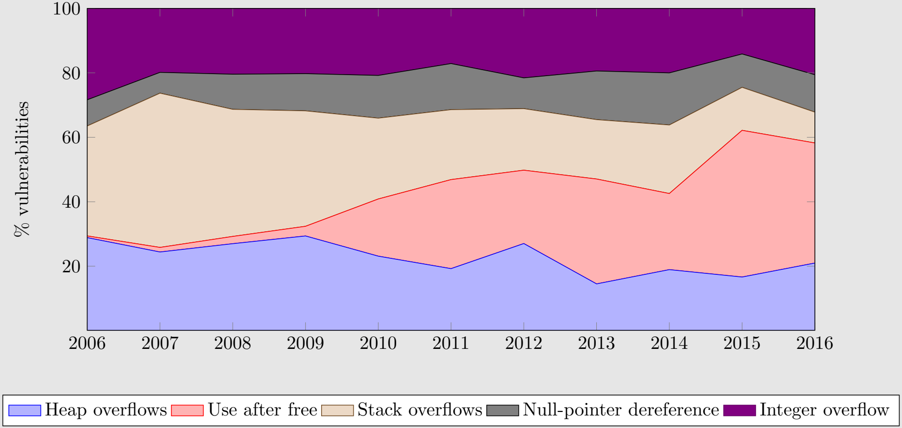
Why are we doing this?

The state reflected by the Common Vulnerabilities and Exposures database.

- Memory errors account for the majority of the critical vulnerabilities
- Large security implications
  - Arbitrary code execution (CVE-2013-1767, CVE-2015-0085, CVE-2016-0937)
  - Leakage of secrets (CVE-2015-7945, CVE-2016-0777, CVE-2014-0160)
- No sign of slowing down

Motivation

Trends in memory errors



Memory Errors

Types

- Spatial: read/write out of bounds

```
int array[10];
/* ... */
array[10] = 10; /* out of bounds write */
```

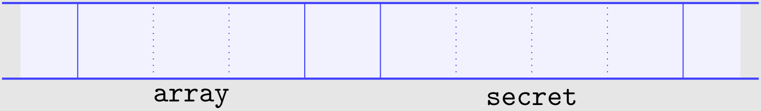
- Temporal: read/write after deallocation

```
int* array = malloc(10 * sizeof(*array));
/* ... */
free(array);         /* deallocate array */
/* ... */
int i = array[0];    /* use after free on read */
```

The Problem

What the attacker does

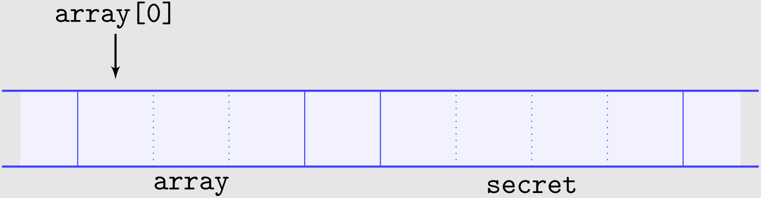
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

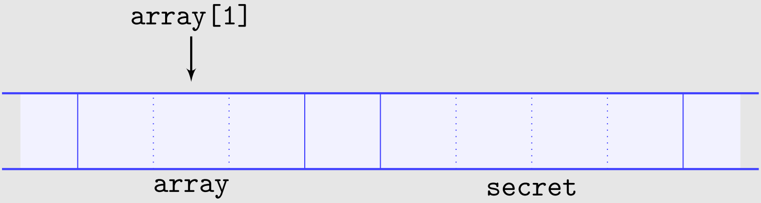
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

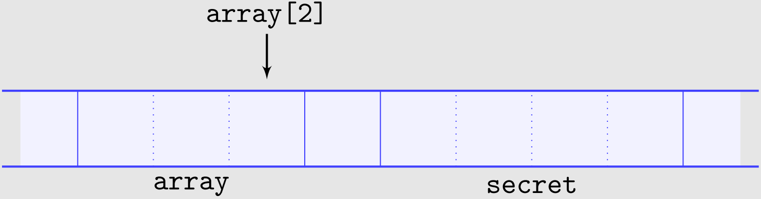
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

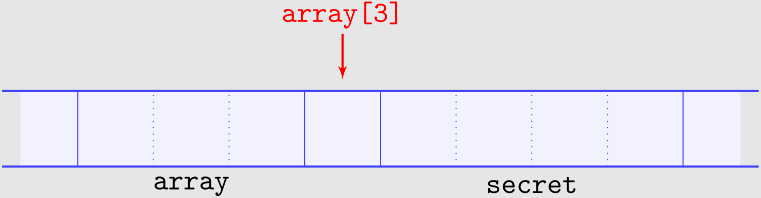
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

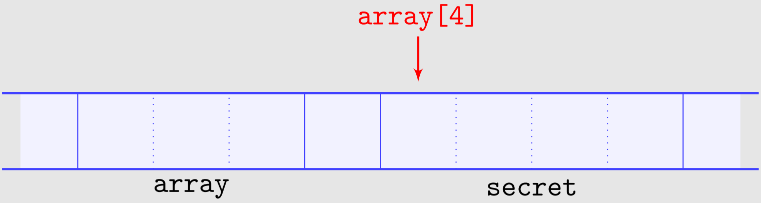
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

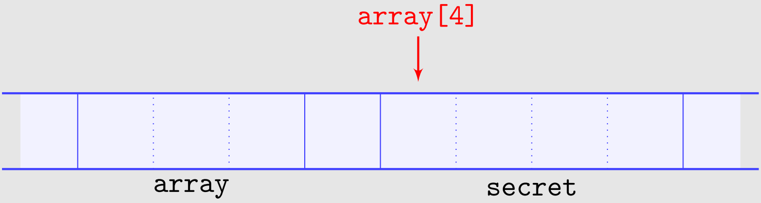
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



The Problem

What the attacker does

```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



Attacker has access to secret!

The Problem

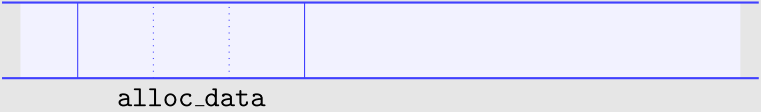
What the attacker does

```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```

The Problem

What the attacker does

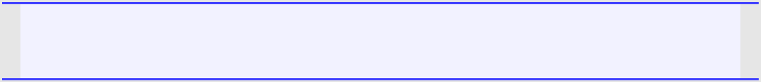
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



The Problem

What the attacker does

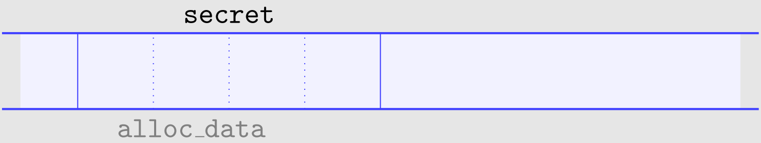
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```


alloc_data

The Problem

What the attacker does

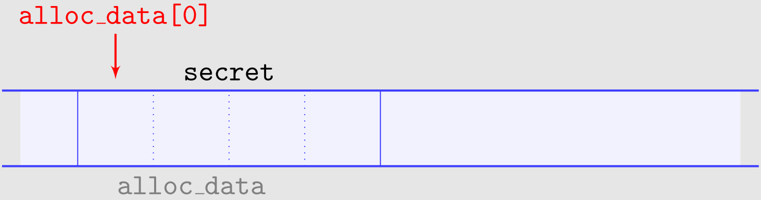
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



The Problem

What the attacker does

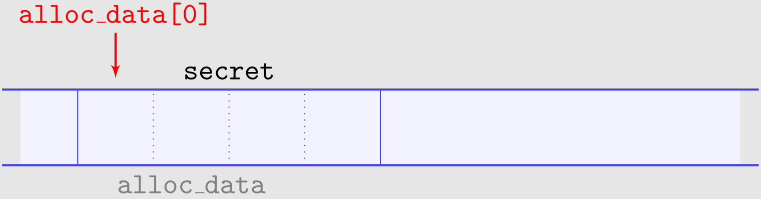
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



The Problem

What the attacker does

```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Attacker has access to secret!

Solutions?

Previous Work

Proposed Method	CT	RT	TE	SE	PO
Baggy Bounds Checking	○	○	○	●	60% [†]
AddressSanitizer	○	○	○	●	73% [‡]
VTPin	●	●	◍	○	17% [‡]
WatchdogLite	○	○	●	●	29% [‡]
Intel MPX	○	○	○	●	n/a
CHERI	○	○	○	●	0% – 15% [††]

[†] SPEC2000 evaluated.

[‡] SPEC2006 evaluated.

[††] Microbenchmarks.

CT Compile time defense, **RT** Run time defense,
TE Temporal error handling, **SE** Spatial error handling, **PO** Performance overhead

Solutions?

Previous Work

Proposed Method	**CT**	**RT**	**TE**	**SE**	**PO**
Baggy Bounds Checking	○	○	○	●	60% [†]
AddressSanitizer	○	○	○	●	73% [‡]
VTPin	●	●	◐	○	17% [‡]
WatchdogLite	○	○	●	●	29% [‡]
Intel MPX	○	○	○	●	n/a
CHERI	○	○	○	●	0% – 15% [††]

[†] SPEC2000 evaluated.

[‡] SPEC2006 evaluated.

[††] Microbenchmarks.

CT Compile time defense, **RT** Run time defense,
TE Temporal error handling, **SE** Spatial error handling, **PO** Performance overhead

Solutions?

Previous Work

Proposed Method	**CT**	**RT**	**TE**	**SE**	**PO**
Baggy Bounds Checking	○	○	○	●	60% [†]
AddressSanitizer	○	○	○	●	73% [‡]
VTPin	●	●	◐	○	17% [‡]
WatchdogLite	○	○	●	●	29% [‡]
Intel MPX	○	○	○	●	n/a
CHERI	○	○	○	●	0% – 15% [††]

[†] SPEC2000 evaluated.

[‡] SPEC2006 evaluated.

[††] Microbenchmarks.

CT Compile time defense, **RT** Run time defense,
TE Temporal error handling, **SE** Spatial error handling, **PO** Performance overhead

Limitations

Why are memory errors still a problem?

- Completeness of the defense
- Completeness of analysis
- Compiler analysis is static, attacks are runtime
- Source code must be available for compiler-based approaches
- Performance overhead

Introducing HA^{2}lloc

Observation

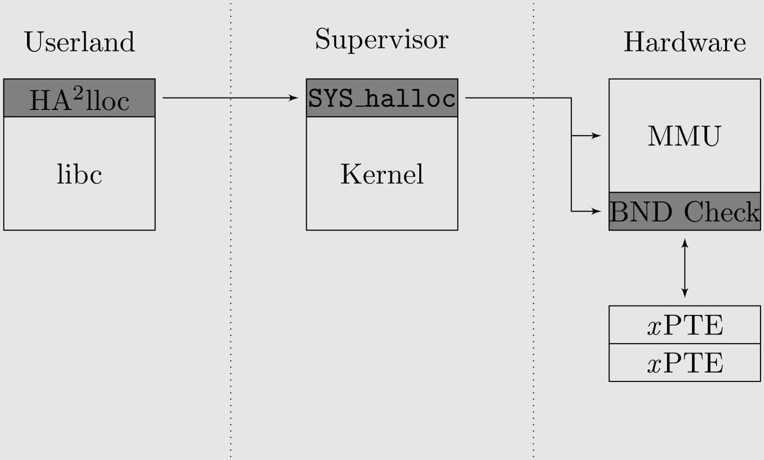
- Allocation size and location is always known at runtime
- Allocator knows when application frees memory

Goals

- Provide heap buffer protection
- Handle both temporal and spatial errors
- Compatible with legacy applications
- Reduce hits in performance

HA^{2}lloc Components

High Level Overview

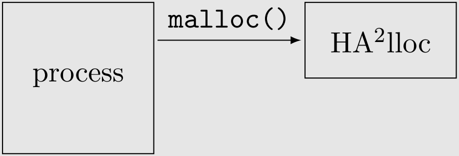


HA^{2}lloc's Allocator

On Allocation

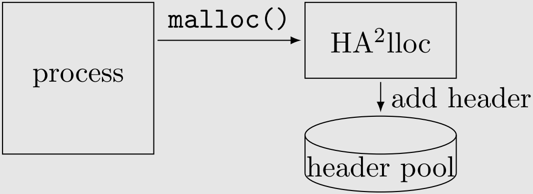
HA^{2}lloc's Allocator

On Allocation



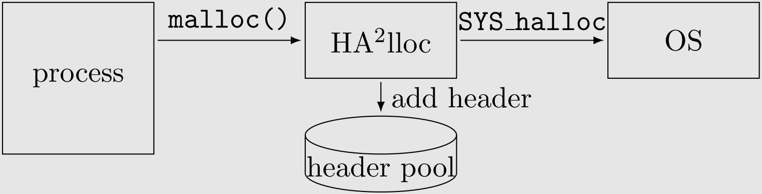
HA^{2}lloc's Allocator

On Allocation



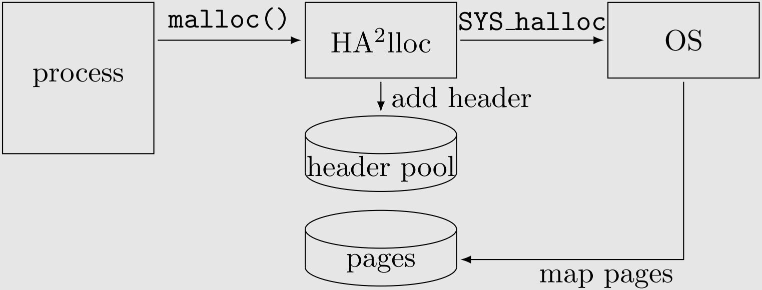
HA^{2}lloc's Allocator

On Allocation



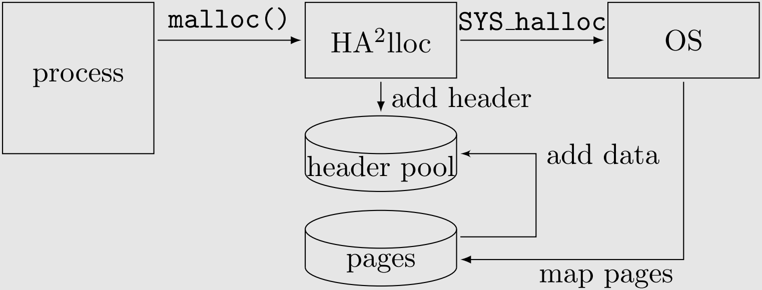
HA^{2}lloc's Allocator

On Allocation



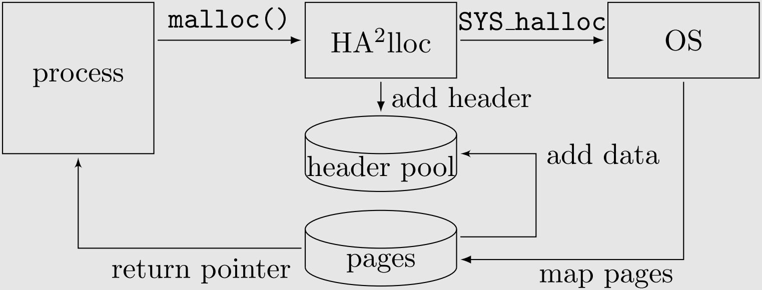
HA^{2}lloc's Allocator

On Allocation



HA^{2}lloc's Allocator

On Allocation



HA^{2}lloc's Allocator

Allocation constrains

- Allocator must take into account alignment requirements
- Type information is lost at compile time
- Must provide an aligment for a worst case scenario
  We allocate on 16 byte boundaries (256 starting points on a 4K page)

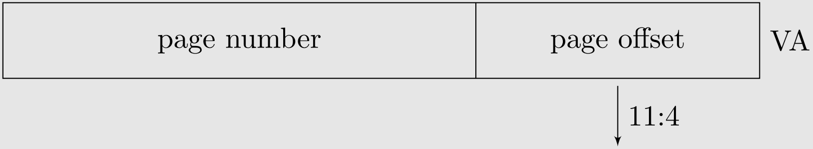
HA^{2}lloc's Allocator

On Access

page number	page offset	VA

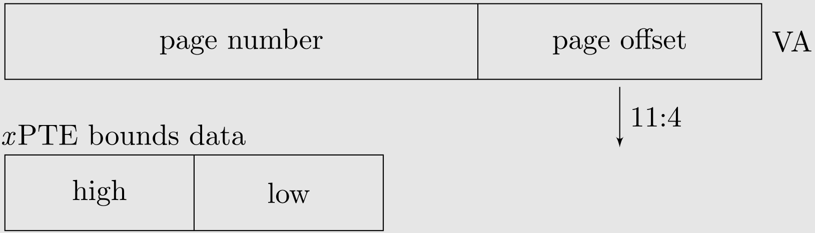
HA^{2}lloc's Allocator

On Access



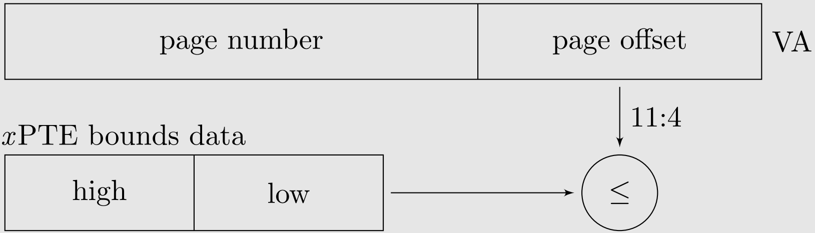
HA^{2}lloc's Allocator

On Access



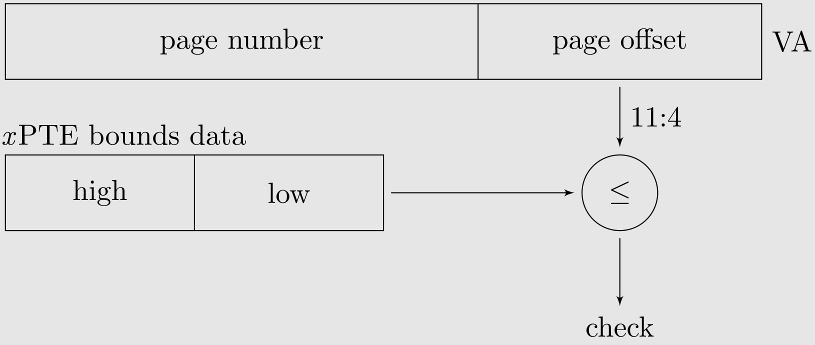
HA^{2}lloc's Allocator

On Access



HA^{2}lloc's Allocator

On Access



page number

page offset

VA

11:4

xPTE bounds data

high

low

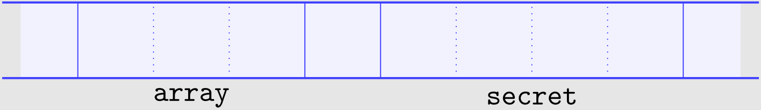
$\leq$

check

Spatial Errors

Back to the old code

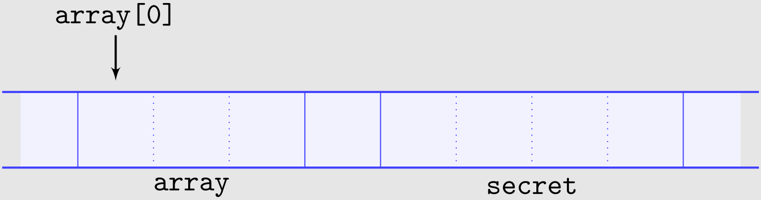
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



Spatial Errors

Back to the old code

```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```

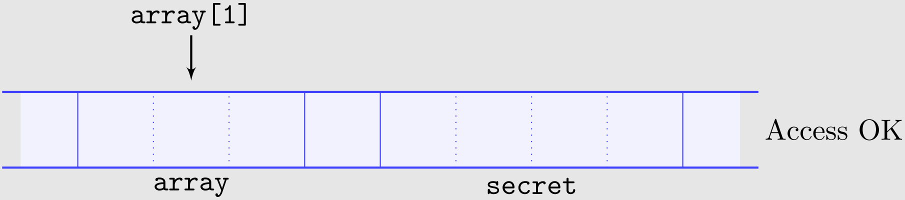


Access OK

Spatial Errors

Back to the old code

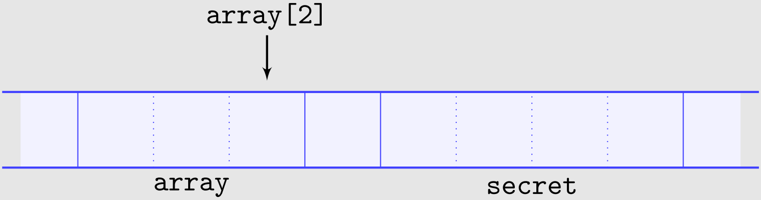
```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



Spatial Errors

Back to the old code

```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```

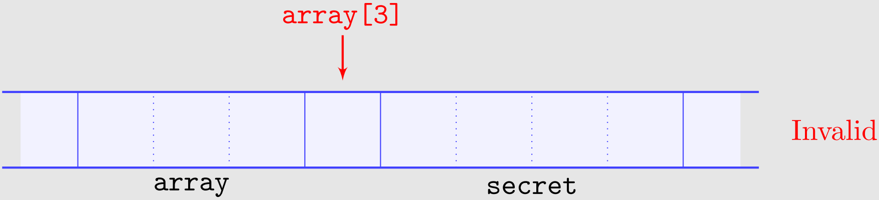


Access OK

Spatial Errors

Back to the old code

```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```

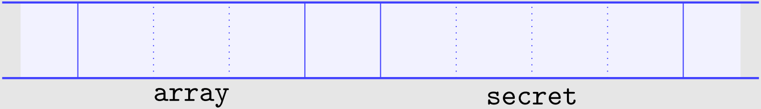


Application receives SIGSEGV

Spatial Errors

Back to the old code

```
int array[3];
int secret[4];
/* ... */
for(size_t i = 0; i < top; i++) {
    transmit(array[i]);
}
```



Invalid

Attacker can not access secret!

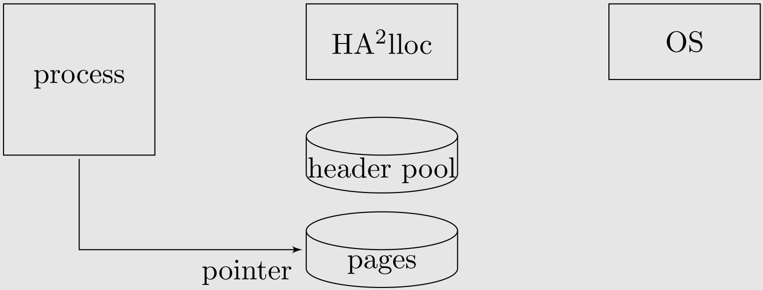
Spatial Errors

Imperative that

- The heap must be randomized
  - Accomplished by `SYS_halloc`
  - Bounds forwarded syscall too
- Allocations must exhibit some form of *redzones* around them
  - Heal alignment requirements and bounds encoding ensure this

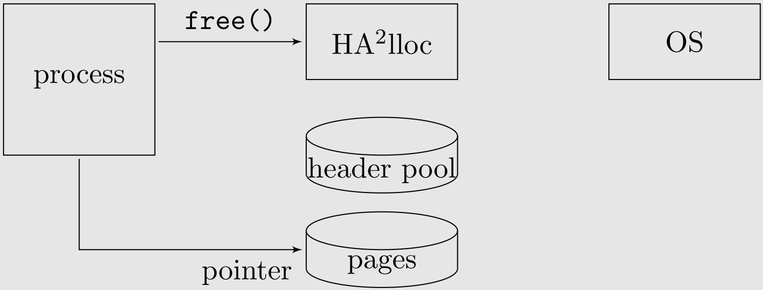
HA^{2}lloc's Allocator

On Deallocation



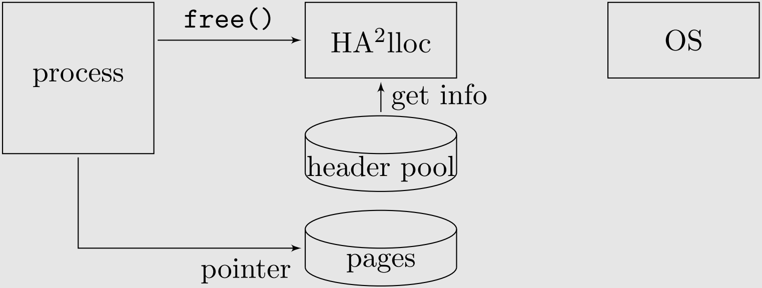
HA^{2}lloc's Allocator

On Deallocation



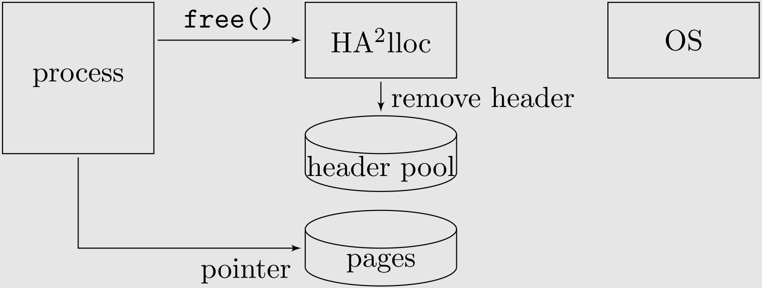
HA^{2}lloc's Allocator

On Deallocation



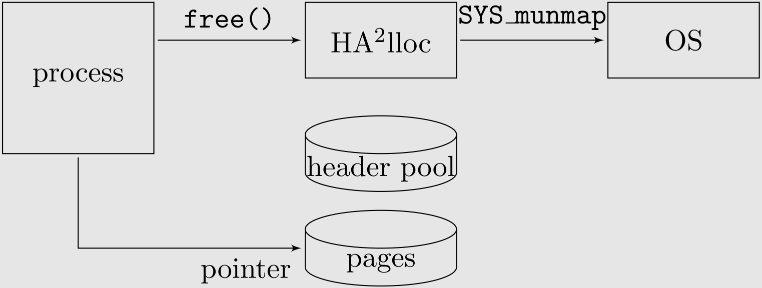
HA^{2}lloc's Allocator

On Deallocation



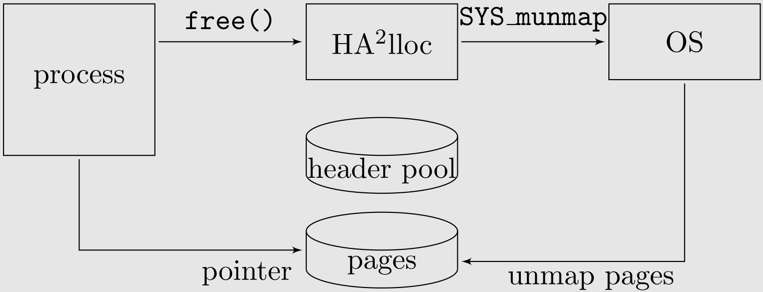
HA^{2}lloc's Allocator

On Deallocation



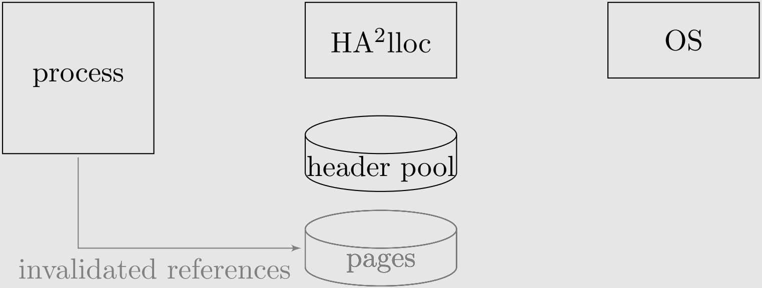
HA^{2}lloc's Allocator

On Deallocation



HA^{2}lloc's Allocator

On Deallocation



Temporal Errors

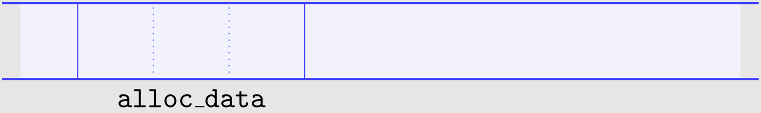
Back to the old code

```c
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```

Temporal Errors

Back to the old code

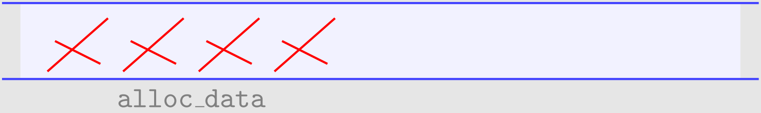
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Temporal Errors

Back to the old code

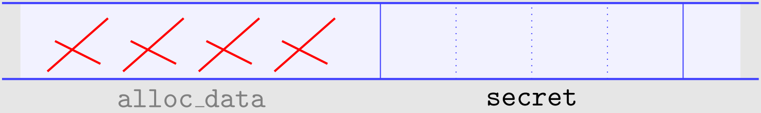
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Temporal Errors

Back to the old code

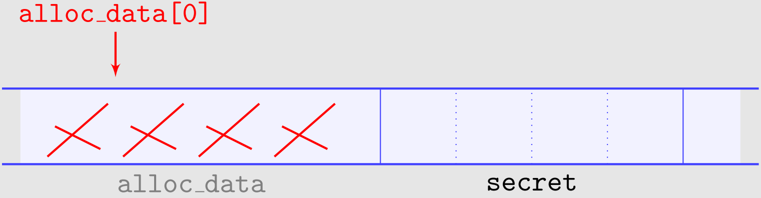
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Temporal Errors

Back to the old code

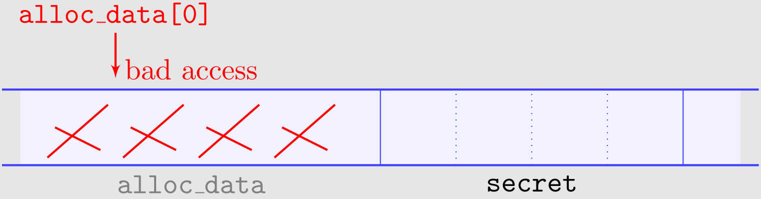
```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Temporal Errors

Back to the old code

```
int* alloc_data = (int*)malloc(sizeof(*alloc_data) * 3);
/* ... */
free(alloc_data);
/* ... */
int* secret = (int*)malloc(sizeof(*secret) * 4);
/* ... */
transmit(alloc_data[0]);
```



Program receives SIGSEGV

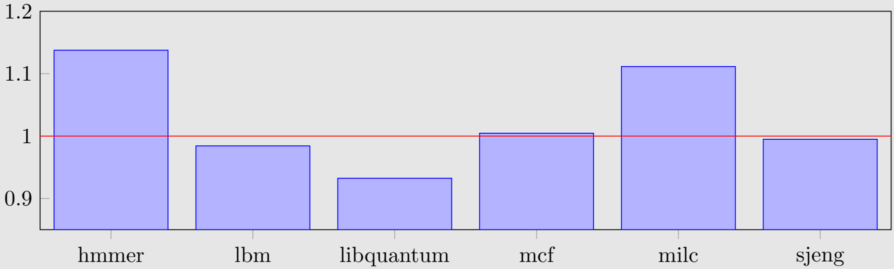
Temporal Errors

Imperative that

- Proper handling of pages with multiple allocations
- Unmapped pages can not be remapped

Evaluation

Performance Evaluation



- We are faster than `glibc`'s allocator for large allocations.
- We are slower than `glibc`'s allocator for small allocations.

Other Works

Comparison to other works

Proposed Method	CT	RT	TE	SE	PO
Baggy Bounds Checking	○	○	○	●	60%
AddressSanitizer	○	○	○	●	73%
VTPin	●	●	◐	○	17%
WatchdogLite	○	○	●	●	29%
Intel MPX	○	○	○	●	n/a
CHERI	○	○	○	●	0% – 15%
Our approach	●	●	●	●	2.5% [†]

[†] Tentative results.

CT Compile time defense, **RT** Run time defense,
TE Temporal error handling, **SE** Spatial error handling, **PO** Performance overhead

Conclusion and Future Work

Conclusion

- Memory errors are still relevant.
- Instrumentation-based approaches have issues.
- Bounds check can be done at runtime with minor overhead.

Moving forward

- Implement hardware component.
  LEON3? Microarchitecture simulator?
- Further testing against actual attacks.

Thank you!

Questions?